Pmbench: A Micro-Benchmark for Profiling Paging Performance on a System with Low-Latency SSDs

Jisoo Yang
jisoo.yang@unlv.edu
Julian Seymour
seymou12@unlv.nevada.edu
Department of Computer Science
University of Nevada, Las Vegas

***Abstract*— Modern non-volatile memory storage devices operate significantly faster than traditional rotating disk media. Disk paging, though never intended for use as an active memory displacement scheme, may be viable as a cost-efficient cache between main memory and sufficiently fast secondary storage.**

However, existing benchmarks are not designed to accurately measure the microsecond-level latencies at which next-generation storage devices are expected to perform. Furthermore, full exploitation of disk paging to fast storage media will require considerations in the design of operating system paging algorithms.

This paper presents *pmbench* – a multiplatform synthetic micro-benchmark that profiles system paging characteristics by accurately measuring the latency of paging-related memory access operations. Also presented are sample pmbench results on Linux and Windows using a consumer NAND-based SSD and a prototype low-latency SSD as swap devices. These results implicate operating system-induced software overhead as a major bottleneck for paging, which intensifies as SSD latencies decrease.

Keywords Pmbench, Paging performance, SSD Swap

1. INTRODUCTION

Pmbench is a user-level micro-benchmark designed to profile system paging performance by measuring memory access latency during fault-intensive memory operations. It is intended to serve as a tool for system architects to diagnose and address issues with paging subsystem. Designed with low-latency SSDs in mind, pmbench aims to accurately measure sub-microsecond intervals while incurring minimal overhead.

Pmbench profiles a system's paging performance by collecting memory access latencies while systematically consuming large quantities of memory to induce paging activity. It measures the time taken for each memory access and compiles the results by keeping a histogram of the measured latencies. Pmbench is carefully written to minimize unnecessary paging activity by using compact, page-aligned data structures and code segment layout.

Pmbench supports Linux and Windows and could be ported to other POSIX-compliant operating systems. It is launched from the command line, creating one or more threads to conduct paging latency measurement based on user-provided parameters. Upon completion, it generates an XML report, which can be read with a companion GUI tool to visually graph and compare data points between benchmarks and further produce a condensed comma-separated report.

We used pmbench to compare the paging performance of Linux and Windows under a heavy paging load using a NAND-based SSD and an experimental low-latency SSD as swap devices. We confirmed that NAND SSDs are entirely unsuitable for DRAM displacement of workloads with working sets exceeding available physical memory. We also found that with low-latency SSD both operating systems suffer from significant software overhead, though Linux paging fares far better than that of Windows.

The rest of this paper is organized as follows: Section 2 presents background on SSDs and system paging. Section 3 motivates pmbench's design and current implementation. Section 4 presents pmbench measurements performed on Linux and Windows using low-latency SSDs. Section 5 discusses related works. We conclude in Section 6.

2. BACKGROUND

Solid-State Drives (SSDs) store data electronically in Non-Volatile Memory (NVM) without moving parts, substantially improving bandwidth and latency. SSDs are designed to fill the role occupied by hard drives and thus retain the same storage *block interface*, consisting of a flat address space of fixed-size blocks, whose size typically ranges from 512 to 4K bytes. The host computer reads and writes on a per-block basis, expecting the device to fetch or update the entire block. Almost all storage I/O standards operate under this block interface.

SSDs are equipped with a controller to manage their NVM specifics while retaining the block interface. For example, NAND Flash, used by most of todays SSDs, does not allow in-place writing to a block. Instead, writes must be done on erased blocks, and erasure operations can only be performed in granularities larger than a block. NAND SSD controllers address this issue by implementing an indirection layer that maps a host-visible block address to an internal Flash address. This layer creates the illusion of updating a block in-place, when in fact it writes to a previously erased Flash, and queues the old Flash as invalid and subject to erasure.

Future SSDs equipped with next-generation NVM will outperform current NAND Flash-based SSDs not only by having faster access latencies, but also by supporting in-place update, which significantly reduces overhead from the SSD controller. Therefore, a *low-latency SSD* capable of delivering a 4KiB block within a couple of microseconds in a sustained load is entirely plausible.

SSDs can thus improve the performance of system *paging*, by which an OS transfers the contents of fixed-size memory pages to/from a storage device. Paging implements virtual memory, enabling execution of programs whose collective working set exceeds the size of available physical memory. The storage device used for paging is also known as a *swap device*.

Page fault is the processor protection mechanism allowing operating systems to implement paging. A page fault is triggered when a processor accesses a virtual address that translates to a memory page not physically mapped. Once triggered, the processor invokes the OS's page fault handler, which ensures the page is loaded into memory, updates the corresponding page mapping, then resumes the instruction that triggered the fault.

If the fault address refers to a page which hasn't been mapped, but the content of the page is still in memory, the OS can handle it without triggering storage I/O. This type of fault is known as *soft* or *minor fault* and can be handled very quickly. If the content of the page cannot be found in memory but is instead located on the swap device, the OS must schedule storage I/O to bring in the data. This is known as a *hard* or *major fault* and can take a very long time to resolve.

3. DESIGN AND IMPLEMENTATION

Pmbench is motivated to profile OS paging performance when a low-latency SSD equipped with next-gen NVM is used as the swap device. We believe, and have confirmed, that current operating systems are inefficient when paging to such devices. As paging was never expected to be fast, its implementation has remained unoptimized, as is apparent when the system swaps to low-latency SSDs. Detailed quantitative analyses of paging must follow to identify and address the inefficiencies inhibiting paging performance.

Pmbench profiles OS paging performance by illustrating the distribution of user-perceived latencies of memory accesses that result in page faults. This requires the systematic consumption of a large quantity of memory in an artificially-generated memory access pattern. Pmbench measures the time taken for each memory access and compiles the results by keeping population counts for latencies measured.

Pmbench's primary technical concern is minimizing overhead from internal memory use. Note that this requirement precludes the use of a trace-based benchmark or recording all measurement data, which would require substantial use of memory and storage I/O, significantly degrading measurement under heavy paging. Although it is impossible to completely prevent interference, efforts were made to minimize memory loads incurred by measurement infrastructure.

The narrow timescale that pmbench tries to accurately measure is another challenge. Events with microsecond-level latency resolution are of interest in profiling operating system paging performance with fast SSDs. In modern systems, a few microseconds translates to tens of thousands of cycles, necessitating close attention to the measurement methods and delicate instruction-level tuning.

Pmbench generates rich access latency statistics in the form of an XML file, which the companion graphical data analysis tool can import, manage, and use to generate graphs visualizing the latency histograms. This comparison can then be exported to a Comma-Separated Value (CSV) file for further processing/analysis with external applications.

Pmbench is executed from the command line with a variety of optional parameters, summarized in Table 1. Pmbench uses a processor timestamp counter to measure the time taken to access an address. The page frame number of each access is determined by a user-selected spatial memory access pattern, each mimicking various statistical distributions; currently the uniform (true random), normal (bell-shaped) and Zipf (long-tail) distributions are supported, along with a deterministic fixed-stride pattern with wraparound. Page offsets are randomly selected with uniform distribution, or can be set to a user-provided fixed value.

Option (Keyword)	Value	Description
Map size (`-m`)	integer	The size of the memory map in mebibytes.
Set size (`-s`)	integer	The portion of the memory map in mebibytes in which accesses will be performed.
Access pattern (`-p`)	"uniform" "normal" "linear"	Access pages in a uniform random fashion (default). Access random pages with normal distribution. Access pages in deterministic fashion with fixed stride.
Shape of pattern (`-e`)	float	Optional parameter determining the shape of the chosen access pattern's distribution.
Read/write ratio (`-r`)	0–100	Percentage of accesses that will be reads. Default is 50.
Number of threads (`-j`)	integer	The number of measurement threads.
Timestamp method (`-t`)	"rdtscp" "rdtsc" "perfc"	Use x86's `rdtscp` instruction (default). Use x86's `rdtsc` instruction. Use OS-provided timestamp method.
Delay (`-d`)	integer	Delay between accesses in clock cycles. Default is 0.
Offset (`-o`)	integer	Offset into the page being accessed. Specifying -1 (default) results in uniform random offset.
Cold (`-c`)	boolean	Skips the warm-up phase of benchmark.
Initialize (`-i`)	boolean	Randomly initialize the memory map.
File name (`-f`)	string	File name for XML output file.

Table 1. Command-line options of pmbench. The last argument of the command line specifies the desired duration of the benchmark in seconds. The Initialize `-i` option was useful in circumventing memory compression newly introduced in Windows 10 [8].

Access instructions, hand-written in assembly to ensure precision, are bookended by a pair of timestamps, whose difference in values will measure in clock cycles the time taken for the memory operation to complete. If the accessed address is in main memory and already mapped in its address space, the reference will take only a few clock cycles, typically resulting in less than 0.5 µs. If an access results in either a minor or major fault, the timestamp difference will be the fault handling latency. The measured latency is then converted to nanoseconds and used to increment a per-thread occurrence counter. The occurrence counter array is constructed in such a way that it first divides intervals into buckets of latencies log base-2. Buckets of lower band (between 2^8 and 2^{23} ns) are further divided into 16 sub-bands with linear divisions[1].

This tiered method of accounting allows pmbench to measure a wide range of latencies, from submicrosecond to thousands of milliseconds, while preserving a detailed profile of smaller samples within sub-10 microsecond intervals where next-gen SSD devices are expected to operate. Pmbench supports a user-specified number of worker threads and increments per-thread latency counters. Care was taken to construct the in-memory counter so that each measurement thread requires a single page. Pmbench minimizes its code footprint by avoiding library function calls in its critical path.

There are two types of memory access: "read" – aligned 4-byte load and "write" – aligned 4-byte store. The 4K page dedicated to each thread for counting occurrence is divided into two 2K counter arrays, each counting for reads and writes. On operating systems that implement memory compression, the allocated memory map can be optionally initialized with random bits to circumvent its effects. Additionally, pmbench collects OS-reported memory and paging statistics over the course of a benchmark run.

The companion GUI tool produces line graphs from the histograms of pmbench result XML files. The parameter set for graphing/export is specified with drop-down menus; the radio buttons specify which variable will be used as the basis of comparison in the graph as well as exported CSV files.

4. EVALUATION

To demonstrate its utility, we used pmbench to evaluate the paging characteristics of Linux and Windows using SSDs as swap devices. The devices tested include a conventional NAND-based SSD and a prototype low-latency SSD.

4.1. Experimental setup

All measurements were conducted on a x86 desktop machine with 4.0 GHz quad-core Intel i7-6700K processor and Z170 chipset. The machine is populated with total 16 GiB DRAM truncated to 2 GiB memory by setting OS boot parameters. An SSD is dedicated as the boot drive for both Linux and Windows partitions, each selected upon boot.

[1]For example, a latency measurement of 9,231 ns will increment the 3rd counter in the 8,192 (2^{13}) - 16,384 (2^{14}) ns bucket.

Samsung's 850 EVO 500 GB SSD is connected via SATA 6Gb/s and used as the swap device for NAND SSD measurements. The peak performance of the Samsung SSD is rated at 100 µs latency for 4KiB random reads at queue depth of 1, and 25 µs for random writes, but the SSD buffers write in internal DRAM, thus sustained average write latency is expected to be at least 100 µs. For low-latency SSD measurements, we used a prototype SSD which simulates the performance of low-latency NVM media. This low-latency SSD uses NVM Express protocol on 8 lanes of PCIe Gen 2 bus and is capable of sustained latencies of 5 µs for both 4 KiB random reads and writes at queue depth of 1.

On Linux we used a standard installation of Fedora 23 Workstation using stock 64-bit kernel version 4.3.5 with page clustering disabled. For Windows we used 64-bit Windows 10 Professional with a custom-sized pagefile optimized for interactive applications. On both operating systems, only the SSD being tested was enabled as a swap device.

We ran pmbench with normal process priority on four combinations of OS and SSD. The results seen here each represent the average of 25 identical benchmarks run for 5 minutes with two worker threads accessing 8 GiB of virtual memory in a uniformly random fashion. The memory was randomly initialized and the read/write mixture set at 50%.[2] We ran the benchmark after the OS enters quiescent state from fresh boot. We did not disable background processes/services but ensured no significant activity (e.g., antivirus scan) interfered with measurement.

4.2. Analyzing pmbench results

We illustrate the analysis of pmbench results by using measurement data from Windows 10 with the low-latency SSD as swap device. Figure 1 shows the access latency population count (i.e., histogram) plotted over latency as the X-axis in log-scale. The two graphs represent the same data, but the top graph's Y-axis uses a linear scale whereas the bottom graph uses a logarithmic scale. Read counts (black dash) and write counts (gray solid) are plotted separately, but there is no meaningful difference between them for accesses involving major faults.

As shown by the peak of the graph, the most frequent access latency is at 14.1 µs. This implies that the majority of major faults that go through the frequent-path have software overhead of about 9 µs – almost twice the SSD latency.

[2]Exact command was: `pmbench -m 8192 -s 8192 -j 2 -r 50 -d 0 -o -1 -p uniform -t rdtscp -c -i 300`

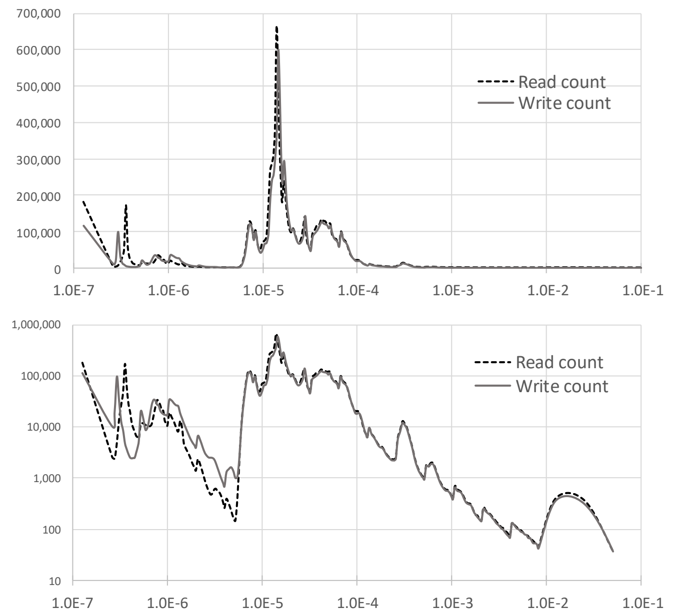

Figure 1. Access latency histogram for Windows 10 with low-latency SSD. The two graphs plot the same data but the bottom graph uses a log scale on Y-axis to highlight the presence of long-latency faults. Both graphs use log scale on X-axis for latency.

Left of the peak are a substantial number of accesses not resulting in page faults. These are the expected 'hits', accessing addresses that just happen to be physically mapped at the time. In our Windows setup around 12% of all accesses result in hits. This is consistent with the fact that around 1 GiB of the 2 GiB system DRAM was available to pmbench after accounting for the kernel's pinned memory.

The histogram exhibits long-tail, meaning there are a substantial number of extremely long-latency faults that heavily increase average latency and therefore hurt overall paging performance. Not surprisingly, the average latency of all major faults is 36.2 µs, far above the 14.1 µs peak latency. The average pure software overhead – latency from operating system processing – is therefore 31.2 µs, 624% of the 5 µs pure hardware latency.

Pmbench's detailed histogram can guide optimization of paging to low-latency SSDs. Measurement data suggests the frequent-path for major faults is too slow. Frequent-path latency could be reduced by eliminating routines unnecessary for low-latency SSDs, and possibly by employing polled I/O instead of interrupt-driven I/O [22]. Furthermore, there are too many long-latency faults: analysis shows that 33% of total execution time

	Swap device (SSD) type:	Low-latency SSD		NAND SSD	
	Operating system:	Linux	Windows	Linux	Windows
1	**Average latency of all accesses including hits:**	7.0 µs	32.6 µs	193 µs	369 µs
2	**Most frequent major-fault latency:**	8.2 µs	14.1 µs	131 µs	385 µs
3	**Average latency of all major-faults:**	8.6 µs	36.2 µs	236 µs	440 µs
4	**Average OS overhead for major-faults:**	3.6 µs	31.2 µs	136 µs	340 µs
5	**OS overhead as percentage of media latency:**	72%	624%	136%	340%

Table 2. Comparison of paging overhead. The last row shows the software overhead of paging as percentage of raw media latency, using 5 µs as media latency for the low-latency SSD and 100 µs for NAND SSD. Refer to Section 4.2 for further explanation of each row.

is spent on faults taking longer than 100 µs to handle, and 12.5% on faults taking 1 ms or more. Redesign of process scheduling policy involving paging in low-memory conditions could address these long-latency faults.

4.3. *Paging performance comparison*

We performed the same analysis on the data measured from all four combinations of OS and SSD, which is summarized in Table 2. Results from NAND SSDs confirm their unsuitability as DRAM displacements for workloads whose working set exceeds available physical memory. Although NAND SSDs can greatly speed up the tasks traditionally performed by HDDs such as file I/O, its fault service latency of 200 µs or more can easily thrash the system. The maximum sustained bandwidth of swap on NAND SSD is merely 20 MiB/sec, which is three orders of magnitude smaller than DRAM bandwidth.

In contrast, the low-latency SSD looks much more promising for use as DRAM displacement, though further OS optimization seems necessary. Measurements show the low-latency SSD delivering performance far superior to the NAND SSD on both OSes, due primarily to substantially lower media access latency. However, the low-latency media demonstrates that a high percentage of the average fault latency can be attributed to inefficiencies of the software component.

It also shows that the current version of Linux handles low-latency SSD paging significantly better than Windows 10: Windows is 8.7 times slower than Linux in processing faults in low-memory conditions. Windows suffers from slow frequent-path and inefficient scheduling policy as discussed in Section 4.2. Linux, though faring better than Windows, still has room for improvement: the average 3.6 µs OS overhead translates to a substantial 14,400 clock cycles. We believe pmbench will prove useful in pinpointing these system bottlenecks.

5. RELATED WORKS

Low-latency SSDs will use next-generation non-volatile memory (NG-NVM) media. Although the incumbent NAND Flash technology continues to achieve higher bit density by going 3D [17], it suffers from inherent latency and management issues preventing DRAM-like usage [10]. NG-NVMs include Spin-Transfer-Torque-RAM [13], [18], which is more suited to replace SRAM cache, and Resistive RAM (ReRAM) [19], which is cost-inefficient for mass market.

Phase-Change Memory (PCM) [16], [20] is the most promising NG-NVM technology at present, and thus the technology which our low-latency SSD was chosen to model. 3D Xpoint memory, recently announced for volume production by Intel and Micron, has performance closely resembling that of PCM.

The emergence of SSDs has renewed interest in storage system optimization and differentiation. The idea of using SSDs for DRAM displacement has also been explored for NAND SSDs via a user-level custom memory allocator [5]. NG-NVM media research has focused on its direct integration onto the memory bus [7], [9], [11].

Virtual memory and demand paging are classic topics covered by most OS textbooks and considered matured. Even the ideas of recoverable, persistent virtual memory [15] and compressed virtual memory [21] are not new. However, the rapidly changing platform landscape makes visiting those ideas worthwhile. Failure-atomic `msync()` [14] for example examines the memory persistence issue in the context of widespread Linux [6] and SSD use as a fast-backing persistent store.

The app-based usage pattern in smartphones and tablets led Windows 10 to introduce memory compression, enabling efficient app-swaps [8]. Commercial adoption of memory compression attests to the feasibility of using low-latency SSDs for DRAM displacement via paging. If done efficiently, paging to low-latency SSDs would likely have similar costs to memory com-

pression; pmbench may assist in optimizing operating systems and platforms for this purpose.

Micro-benchmarks are useful for systems engineers to find inefficiencies and fine-tune with surgical precision. LMBench [3], [12] has long been used to measure performance of OS system calls. For storage systems, FIO [1] and IOMeter [2] are well-known micro-benchmarks that produce detailed storage stack statistics. However, existing benchmarks are not designed to accurately measure paging performance with microsecond-level precision. Pmbench aims to fill that void.

6. CONCLUSION

Low-latency next-generation SSDs are already coming to market. Using them as a fast paging space could be a cost-efficient alternative to large pools of DRAM. However, as we have confirmed using pmbench, this cannot occur until measures are taken to improve the demonstrably inefficient performance of current operating systems when paging to such low-latency devices. Systems developers can use pmbench's detailed paging performance data to locate inefficiencies at the source and explore possible solutions.

In the future we intend to improve pmbench by supporting additional operating systems and synthesizing memory access patterns with greater accuracy by analyzing patterns obtained from real workloads.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by an award from Intel Corporation.

REFERENCES

[1] Flexible I/O tester (fio). `https://github.com/axboe/fio`. Accessed: 2016-10-28.

[2] Iometer. `http://www.iometer.org/`. Accessed: 2016-10-28.

[3] Lmbench source. `https://sourceforge.net/projects/lmbench/`. Accessed: 2016-10-28.

[4] PMBench. `https://bitbucket.org/jisooy/pmbench`. Accessed: 2016-10-28.

[5] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM memory management made easy. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, March 2011.

[6] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, 2005.

[7] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, United Kingdom, March 2012.

[8] Ethan Creeger. Windows 10: Memory compression. `https://riverar.github.io/insiderhubcontent/memory_compression.html`. Accessed: 2016-10-28.

[9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*, Amsterdam, The Netherlands, 2014.

[10] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND Flash memory. In *Proceedings of the 2012 USENIX/ACM Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2012.

[11] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, June 2009.

[12] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996.

[13] Asit K. Mishra, Xiangyu Dong, Guangyu Sun, Yuan Xie, N. Vijaykrishnan, and Chita R. Das. Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA'11)*, San Jose, CA, June 2011.

[14] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic `msync()`: A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, Prague, Czech Republic, April 2013.

[15] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP'93)*, Asheville, NC, December 1993.

[16] DerChang Kau *et al*. A stackable cross point phase change memory. In *Proceedings of the 2009 IEEE International Electron Devices Meeting (IEDM)*, pages 1–4, Baltimore, MD, December 2009.

[17] Jae-Woo Im *et al*. A 128Gb 3b/cell V-NAND Flash memory with 1gb/s I/O rate. In *Proceedings of the 2015 International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, February 2015.

[18] Ki Chul Chun *et al*. A scaling roadmap and performance evaluation of in-plane and perpendicular MTJ based STT-MRAMs for high-density cache memory. *IEEE Journal of Solid-State Circuits*, 48(2):598–610, February 2013.

[19] Richard Fackenthal *et al*. A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *Proceedings of the 2014 International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, February 2014.

[20] Youngdon Choi *et al*. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Proceedings of the 2012 International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, February 2012.

[21] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[22] Jisoo Yang, David Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 2012 USENIX/ACM File and Storage Technology (FAST)*, Santa Clara, CA, February 2012.